Assignment 3

Problem 1 (10.1-2)

Basic Idea: We can make it similar to the heap and stack in the computer. That is, starting at the edge and forwarding to the middle.

```
template <class T>
class twoStack {
    private:
        T array[n] = {}; // replace n with an actual number
        size_t stack1Top = 0, stack2Top = n-1;

    public:
    /* constructor and destructor */

        void pushStack1(T val) {
            if (stack1Top > stack2Top) {
                throw /* overflow error */
            }
            array[stack1Top++] = val;
        }

        T popStack1() {
            if (stack1Top == 0) {
                throw /* underflow error */
            }
            return array[stack1Top--];
        }

        void pushStack2(T val) {
            if (stack2Top < stack1Top) {
                throw /* overflow error */
            }
            array[stack2Top--] = val;
        }
```

```
        T popStack2() {
            if (stack2Top == n-1) {
                throw /* underflow error */
            }
            return array[stack2Top++];
        }
};
```

Problem 2 (10.1-6)

```
template <class T>
class stackQueue {
    private:
        stack<T> stack1 {};
        stack<T> stack2 {};

    public:
    /* constructor and destructor */

        void push(T val) {
            stack1.push(val);
        }

        T pop() {
            if (this.empty()) {
                throw /* some error */
            }
            if (stack2.empty()) {
                while (!stack1.empty()) {
                    stack2.push(stack1.pop());
                }
            }
            return stack2.pop();
        }

        bool empty() {
            return stack1.empty() && stack2.empty();
        }

        T front() {
            if (this.empty()) {
```

```
                throw /* some error */
            }
            if (stack2.empty()) {
                while (!stack1.empty()) {
                    stack2.push(stack1.pop());
                }
            }
            return stack2.top();
        }

        T back() {
            if (this.empty()) {
                throw /* some error */
            }
            if (stack1.empty()) {
                while (!stack2.empty()) {
                    stack1.push(stack2.pop());
                }
            }
            return stack1.top();
        }

        size_t size() {
            return stack1.size() + stack2.size();
        }
};
```

Time Complexity Analysis

- `push()` : O(1), as the `push()` of stack is O(1)[1] and we call it once.
- `pop()` : O(n), as the `pop()` of stack is O(1)[2] and we may call up to n times in the while loop (`pop()` after `push()` n elements).
- `empty()` & `size()` : O(1), as the `empty()` and `size()` of stack is O(1)[3][4] and we call it twice.
- `front()` & `back()` : O(n), it is similar with `pop()` . Although the `top()` of stack is O(1)[5], we may call up to n times in the while loop.

Problem 3 (10.2-5)

```
/* Definition of the Node */
template <class T>
class dic {
    private:
        Node* head = nullptr;

    public:
    /* constructor and destructor */

        void insert(T t) {
            if (head == nullptr) {
                head = new Node(t);
                head->next = head;
            } else {
                head->next = new Node(t, head->next);
            }
        }

        void remove(T t) {
            auto ptr = head;
            while (ptr->next->val != t) {
                ptr = ptr->next;
                if (ptr == head) {
                    throw /* some error */
                }
            }
            auto toRm = ptr->next;
            if (toRm == ptr) {
                head = nullptr;
            } else {
                ptr->next = toRm->next;
            }
            delete toRm;
        }

        T* search(T t) {
            auto ptr = head;
            while (ptr->val != t) {
                ptr = ptr->next;
```

```
                if (ptr == head) {
                    return /* something means cannot find */
                }
            }
            return ptr;
        }
};
```

Time Complexity Analysis

- `insert()`: takes O(1), we just need to construct a new `Node`, no matter with the number of elements.
- `remove()`: takes O(n), we may need to traverse the whole circular in the loop to find the target.
- `search()`: takes O(n), we may need to traverse the whole circular in the loop to find the target.

Problem 4 (12.2-7)

- `TREE-MINIMUM()` always goes to the left child until it is `null`. It depends on the height of the tree. As the height of BST must be $\lfloor \lg n \rfloor$ (proof is similar with Problem 7), it is $\Theta(\lg n)$.
- `TREE-SUCCESSOR()` involves two cases:
    1. It has right child. We take this right child as a tree root to find `TREE-MINIMUM()`.
    2. It has no right child but parent. Check whether this node is the left child. If yes, its parent is the successor. If no, take the parent as the one to check the parent. That is, check parent recursively until we find the node is the left child or the tree root. If the latter case happened, the original node must be the last element.

Then, according to the previous rules, we can point out that we must traverse each edge of the tree twice. As one edge means one move of the pointer (go up or down), the number of edges makes the conclusive result of time complexity. The edges of BST with n vertices is $n - 1$. Therefore, the time complexity should be $\Theta(n)$.

Problem 5 (12.3-4)

No.

```
// delete 4 first, then delete 3.
   4           6           6
  / \         / \           \
 3   7  ==>  3   7  ==>      7
    /
   6

// delete 3 first, then delete 4.
   4           7           7
  / \         / \           \
 3   7  ==>  4   6  ==>      6
    /
   6
```

Problem 6 (12.3-5)

Use AVL Tree as the BST.

```
/* Definition of Node */
template <class T>
class tree {
    private:
        Node* root; // always point to the tree root

        // rotations does not make influence on linear sequence
        Node* rightRotate(Node* root) {
            auto x = root->left;
            root->left = x->right;
            x->right = root;
            return x;
        }

        Node* leftRotate(Node* root) {
            auto x = root->right;
            root->right = x->left;
            x->left = root;
            return x;
```

```
        }

        long long getHeight(Node* n) {
            if (n == nullptr) {
                return 0;
            }
            return n->height;
        }

        Node* search_helper(T t, Node* root) {
            if (root == nullptr) {
                return nullptr; // means not exist
            } else if (t == root->val) {
                return root;
            } else if (t < root->val) {
                return search_helper(t, root->left);
            } else {
                return search_helper(t, root->right);
            }
        }

        // lastRight means the one need to change its succ
        // lastLeft means the succ of the new Node
        Node* insert_helper(T t, Node* root, Node* lastRight,
Node* lastLeft) {
            if (root == nullptr) {
                Node* res = new Node(t, lastLeft); // Node(T val,
Node* succ)
                if (lastRight != nullptr) {
                    lastRight->succ = res;
                }
                return res;
            }
            auto res = root;
            if (t < res->val){
                res->left = insert_helper(t, res->left, lastRight,
res);
            } else {
                res->right = insert_helper(t, res->right, res,
lastLeft);
            }
```

```
            auto balance = getHeight(res->left) - getHeight(res->right);
            if (balance > 1) {
                if (t < res->left->val) {
                    res = rightRotate(res);
                } else {
                    res->left = leftRotate(res->left);
                    res = rightRotate(res)
                }
            } else if (balance < -1) {
                if (t > res->right->val) {
                    res = leftRotate(res);
                } else {
                    res->right = rightRotate(res->right);
                    res = leftRotate(res);
                }
            }
            res->height = max(getHeight(res->left), getHeight(res->right)) + 1;
            return res;
        }

        // lastRight means the one need to change its succ
        // lastLeft means the the replacement of the deleted Node in succ of lastRight
        Node* insert_helper(T t, Node* root, Node* lastRight, Node* lastLeft) {
            if (root == nullptr) {
                throw /* some error */
            }
            if (root->val == t) {
                delete root;
                if (lastRight != nullptr) {
                    lastRight->succ = lastLeft;
                }
                return nullptr;
            }
            auto res = root;
            if (t < res->val){
                res->left = insert_helper(t, res->left, lastRight, res);
```

```
            } else {
                res->right = insert_helper(t, res->right, res,
lastLeft);
            }
            auto balance = getHeight(res->left) - getHeight(res-
>right);
            if (balance > 1) {
                if (t < res->left->val) {
                    res = rightRotate(res);
                } else {
                    res->left = leftRotate(res->left);
                    res = rightRotate(res)
                }
            } else if (balance < -1) {
                if (t > res->right->val) {
                    res = leftRotate(res);
                } else {
                    res->right = rightRotate(res->right);
                    res = leftRotate(res);
                }
            }
            res->height = max(getHeight(res->left), getHeight(res-
>right)) + 1;
            return res;
        }

    public:
    /* constructor and destructor */

        Node* search(T t) {
            return search_helper(t, this.root);
        }

        void insert(T t) {
            this.root = insert_helper(T t, this.root, nullptr,
nullptr);
        }

        void remove(T t) {
            this.root = remove_helper(T t, this.root, nullptr,
nullptr);
```

```
        }
}
```

Problem 7 (6.1-2)

As it is stored as complete binary tree, we can point the relation of height h and the maximum of nodes in this level, which is 2^h. We have:

$$2^0+2^1+2^2+\ldots+2^{h-1}=\frac{1-2^h}{1-2}=2^h-1<n$$

$$2^0+2^1+2^2+\ldots+2^{h}=\frac{1-2^{h+1}}{1-2}=2^{h+1}-1\geq n$$

As n is an positive integer, we have:

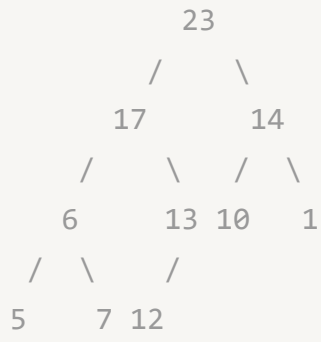
$$2^h\leq n<2^{h+1}$$

$$h\leq \lg n<h+1$$

As h must be an non-negative integer.

$$h=\lfloor \lg n \rfloor$$

Problem 8 (6.1-6)



Obviously, it is not, as 7 > 6.

---

1. Retrieved from cplusplus.com
2. Retrieved from cplusplus.com
3. Retrieved from cplusplus.com
4. Retrieved from cplusplus.com
5. Retrieved from cplusplus.com